EVALUATION OF JAVA AS AN INTERNET DELIVERY TOOL

S. J. Wormley
Center for Nondestructive Evaluation
Iowa State University
Ames, Iowa 50011-3042

INTRODUCTION

The program for Integrated Design, NDE and Manufacturing Sciences at the Center for Nondestructive Evaluation (CNDE) has as its mission to integrate NDE inspectability, materials and stress information, and overall product reliability into the designer's computer aided design capabilities. In recent years the Center has provided design data to give design teams a way to consider the effects of certain design changes on inspectability in typical canonical family geometries of design parts. Providing software modules via the Internet is a logical extension to provide data.

BACKGROUND

Last year several secure Common Gateway Interface (CGI) post-query programs were successfully tested that provide a general structure for allowing the user to specify a number of input parameters, then execute the CGI binary code. The CGI binary code consists of a front-end for reading the user input parameters and the executable software module of any nature. This can be followed by a redisplay of the user window with relevant results, or may spawn display software on the user's computer.

Because making software modules available in this fashion puts a computational burden on CNDE machines, Java is being investigated as an Internet delivery tool. Java code executes on the user's cpu, therefore, minimizing the burden on CNDE machines. Java was designed to be cross-platform in source form like C. Java is compiled to an intermediate byte-code which is interpreted on the fly by the Java interpreter. Thus to port Java programs to a new platform all that is needed is a port of the interpreter. Since the

Review of Progress in Quantitative Nondestructive Evaluation, Vol. 16
Edited by D.O. Thompson and D.E. Chimenti, Plenum Press, New York, 1997

Java interpreter is built into Netscape versions 2.0 and higher, all that is needed is the installation of Netscape on the user's computer.

WHAT IS JAVA

The Java language originated as part of a research project to develop advanced software for a wide variety of networked devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating environment. When the project was started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language environment. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result was a language environment that has proven ideal for development of secure, distributed, network-based end-user applications in environments ranging from networked embedded devices to the World-Wide Web and the desktop.

A primary goal of the Java language was a simple language that could be programmed without extensive programmer training and which would be roughly attuned to current software practice. The Java language was designed as an object-oriented language from the ground up. Object-oriented design, simply stated, is a technique that focuses design on the data, i.e. objects and in the interfaces to it. The needs of distributed, client-server based systems coincide with the packaged, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. The Java language provides a clean and efficient object-based development environment [1].

Java can be characterized as a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language. One way to characterize a system is with a set of buzzwords. Java designers use a standard set of them in describing Java.

Simple

Java designers built a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. Most programmers working these days use C, and most programmers doing object-oriented programming use C++. Even though C++ was unsuitable, Java designers designed Java as closely to C++ as possible in order to make the system more comprehensible. Java omits many rarely used, poorly understood, confusing features of C++ that bring more grief than benefit. These omitted features primarily consist of operator overloading (although the Java language does have method overloading), multiple inheritance, and extensive automatic coercions.

Garbage collection was added thereby simplifying the task of Java programming but making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection the Java language not only makes the programming task easier, it also dramatically cuts down on bugs.

Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.

Object-Oriented

Simply stated, object-oriented design is a technique that focuses design on the data (i.e. objects) and on the interfaces to it. To make an analogy with ultrasonic scanning, an

"object-oriented" scanner would be mostly concerned with the part he was building, and secondarily with the scanning system used to make the scan; a "non-object-oriented" scanner would think primarily of his scanning tank and software. Object-oriented design is also the mechanism for defining how modules "plug and play." The object-oriented facilities of Java are essentially those of C++, with extensions from Objective C for more dynamic method resolution.

Distributed

Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the net via URLs with the same ease that programmers are used to when accessing a local file system.

Robust

Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error prone. One of the advantages of a strongly typed language (like C++) is that it allows extensive compile-time checking so bugs can be found early. Unfortunately, C++ inherits a number of loopholes in compile-time checking from C, which is relatively lax (particularly method/procedure declarations). In Java, declarations are required and do not support C-style implicit declarations. The linker understands the type system and repeats many of the type checks done by the compiler to guard against version mismatch problems.

The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting. While Java doesn't make the QA problem go away, it does make it significantly easier.

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons is that they are very robust: you don't have to worry about freeing or corrupting memory. Programmers can be relatively fearless about dealing with memory because they don't have to worry about it getting corrupted. Java has this property and it has been found to be very liberating. One reason that dynamic languages are good for prototyping is that they don't require you to pin down decisions early on. Java has exactly the opposite property; it forces you to make choices explicitly. Along with these choices come a lot of assistance: you can write method invocations and if you get something wrong, you are informed about it at compile time. You don't have to worry about method invocation error. You can also get a lot of flexibility by using interfaces instead of classes.

Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption. There is a strong interplay between "rcbust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do have access to. This closes the door on most activities of viruses.

Architecture Neutral

Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system

architectures. To enable a Java application to execute anywhere on the network, the compiler generates an architecture neutral object file format -- the compiled code is executable on many processors, given the presence of the Java runtime system. This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh. With the PC market (through Windows/NT) diversifying into many CPU architectures, and Apple moving off the 68000 towards the PowerPC, this makes the production of software that runs on all platforms almost impossible. With Java, the same version of the application runs on all platforms. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

Portable

Being architecture neutral is a big chunk of being portable, but there's more to it than that. Unlike C and C++, there are no "implementation dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point number. Making these choices is feasible in this day and age because essentially all interesting CPU's share these characteristics. The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows and the Macintosh. The Java system itself is quite portable. The new compiler is written in Java and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.

Interpreted

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory. As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on, and what the RPC protocol derivation is based on. It also makes programs more amenable to debugging.

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. For those accustomed to the normal design of a compiler and dynamic loader, this is somewhat like putting the final machine code generator in the dynamic loader. The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple. Reasonably good code is produced: it does automatic register allocation and the compiler does some optimization when it produces the bytecodes. In interpreted code about 300,000 method calls per second is achieved on an Sun Microsystems SPARCStation 10. The performance of bytecodes converted to machine code is almost indistinguishable from native C or C++.

Multithreaded

There are many things going on at the same time in the world around us. Multithreading is a way of building applications with multiple threads. Unfortunately, writing programs that deal with many things happening at once can be much more difficult

than writing in the conventional single-threaded C and C++ style. Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm that was introduced by C.A.R.Hoare. By integrating these concepts into the language they become much easier to use and are more robust. Much of the style of this integration came from Xerox's Cedar/Mesa system. Other benefits of multithreading are better interactive responsiveness and real-time behavior. This is limited, however, by the underlying platform: stand-alone Java runtime environments have good real-time behavior. Running on top of other systems like Unix, Windows, the Macintosh, or Windows NT limits the real-time responsiveness to that of the underlying system.

Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. For example, one major problem with using C++ in a production environment is a side-effect of the way that code is always implemented. If company A produces a class library (a library of plug and play components) and company B buys it and uses it in their product, then if A changes it's library and distributes a new release, B will almost certainly have to recompile and redistribute their own software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) problems can result. Another example, if A distributes an upgrade to its libraries then all of the software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly. By making these interconnections between modules later, Java completely avoids these problems and makes the use of the object-oriented paradigm much more straightforward. Libraries can freely add new methods and instance variables without any effect on their clients.

Java understands interfaces-- a concept borrowed from Objective C which is similar to a class. An interface is simply a specification of a set of methods that an object responds to. It does not include any instance variables or implementations. Interfaces can be multiply-inherited (unlike classes) and they can be used in a more flexible way than the usual rigid class inheritance structure.

Classes have a runtime representation: there is a class named Class, instances of which contain runtime class definitions. If, in a C or C++ program, you have a pointer to an object but you don't know what type of object it is, there is no way to find out. However, in Java, finding out based on the runtime type information is straightforward. Because casts are checked at both compile-time and runtime, you can trust a cast in Java On the other hand in C and C++, the compiler just trusts that you're doing the right thing. It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and have it easily dynamically-linked into the running system.

EVALUATION OF JAVA

Evaluation of Java in the context of the program for Integrated Design, NDE and Manufacturing Sciences at the Center for Nondestructive Evaluation has to answer the question--does it facilitate the application of design related codes to user problems and does it aid in the transfer of technology to industry better that alternative methods? Three ways to run program modules are considered.

1. Port to and install a program module on a user's machine. That code is then operated by the user on the user's computer.

2. Make a program module available via the Internet using a CGI interface with the code running on the provider's computer with input and output being served across the Internet.

3. Translate a program module into the JavaScript, allowing the user to download the Java code via the internet and execute the same on the user's platform.

A Gauss-Hermite beam model code similar to that used in UTSIM is a reasonable candidate for comparisons. The code is large enough and computer intensive enough that comparisons will reflect a real application environment. Work is in progress to port the Gauss-Hermite code to JavaScript and to quantity of network traffic generated, and the effect of heavy network traffic on the user's real time performance. Principal Investigators developing powerful simulation tools such as XRSIM, UTSIM and ECSIM come with the biases in terms of favored programming languages. Part of an overall evaluation of Java must include a measure of the effort involved in converting program code developed in C, C++, or FORTRAN to the Java Language.

SUMMARY

Java's primary advantages are the portability to different user platforms and the fact that code is executed by the user's CPU. There is also the possibility of arbitrary graphics, data types, and greater user interaction. Real tradeoffs in program translation, network traffic and the user's real time performance will be accessed by testing a substantial program module such as the Gauss-Hermite beam model module developed as part of the program for Integrated Design, NDE and Manufacturing Sciences at the Center for Nondestructive Evaluation.

ACKNOWLEDGMENTS

This work was supported by the Center for Nondestructive Evaluation at Iowa State University and is part of the NIST program for Integrated Design, NDE and Manufacturing at Iowa State University.

REFERENCES

1. J. Gosling & H.McGilton, *The Java Language Environment: A White Paper*, http://java.motiv.co.uk/intro/javawhitepaper_1.html, 1996.

2. K. M. Krom & W. R. Jernigan, Frequently Asked Questions (FAQ) for comp.lang.java, http://www.bimel.com.tr/faq/java-faq.html.

3. E. R. Harold, *Java FAQ list and Tutorial: a work in progress*, http://sunsite.unc.edu/javafaq/javafaq.html, March 27, 1996.

4. D. Williams, *Collection of Java Code and Examples*, Graphic Layout Example4, http://www.citylimits.com/users /daverw/java/demo/GraphLayout/example4.html.

5. T. Ritchey, *Programming JavaScript for Netscape 2.0*, New Riders Publishing, Indianapolis, Indiana, 1996.

6. T. Ritchey, *Java!*, New Riders Publishing, Indianapolis, Indiana, 1995.

7. T. Budd, *An Introduction to Object-Oriented Programming*, Addison Wesley Publishing Company, Reading, Massachusetts.

8. C. A. R. Hoare, Monitors: An Operating System Structuring Concept, *Communications of the ACM*, Volume 17 Number 10, 1974, Pages 549-557.

9. J. Gosling & F. Yellin, *The Java Application Programming Interface Volume 1: Core Packages.*

10. J. Gosling & F. Yellin, *The Java Application Programming Interface Volume 2: Window Toolkit and Applets.*

11. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*

12. K. Arnold and J. Gosling, *The Java Programming Language*

13. M. Campione and M. Walrath, *The Java Tutorial: Object-Oriented Programming for the Internet.*